Reflections on designing a Virtual Highway Path Planner

Mithi Sevilla - 1 August 2017

Right now it swerves like a crazy wishy-washy undecided driver but to be honest I'm just happy it navigates towards around the virtual loop several times bumping to anyone or exceeding the maximum speed, acceleration, and jerk requirements.

The first project for the third (and last!) term of Udacity's Self-driving Car Engineer Nanodegree is a path planner. I think so far this is the most difficult project yet, as it isn't as straightforward as our previous projects. This article is about how I approached this project.

The goal is to create a path planning pipeline that would smartly, safely, and comfortably navigate a virtual car around a virtual highway with other traffic. We are given a map of the highway, as well as sensor fusion and localization data about our car and nearby cars and we are supposed to give back a set of points **`(x, y)`** in a map that a perfect controller will execute every 0.02 seconds. Navigating safely and comfortably means we don't bump into other cars, we exceed the maximum speed, acceleration and jerk requirements. Navigating smartly means we change lanes when the car in front of us is too slow.

Outline of Contents

Design Guidelines I followed

"All models are wrong but some models are useful" *- Probably George Box*

"Everything should be made as simple as possible, but not simpler."

- Probably NOT albert Einstein

"YAGNI" *- Extreme programmers*

"If it ain't broke, don't fix it" *//Disclaimer: this does not apply to societal problems*

To be honest, for a long period of time I was in a state of analysis-paralysis and did not even write a single line of code. I was thinking about so many things. There were so many concepts taught to us, I wanted to consider them all. Which data is important and which isn't? How far into the future should I plan? Should I predict what the nearby cars are gonna do before deciding what to do? What method should I use in prediction? How accurate and useful are these predictions? How many possible behaviors should I consider? (slow down, go fast, turn right, prepare to turn left) For any behavior, how many paths should I consider before deciding which is the best one? (If I decide to go slow, how slow? If I decide to turn left when exactly in the left? How do I make sure that my path is the best and does not violate any safety or comfort guidelines?

And on top of that, the pseudo-architect in me was also talking. What data structures should I use? What classes and objects should I define? How will they interact with one another. What are the members responsibilities of each object? Does it even make sense that responsibility belongs to that object?

After some contemplation, I remembered something I read a while back by George Box, a famous statistician.

"Now it would be very remarkable if any system existing in the real world could be exactly represented by any simple model. However, cunningly chosen parsimonious models often do provide remarkably useful approximations. For example, the law PV = RT relating pressure P, volume V and temperature T of an "ideal" gas via a constant R is not exactly true for any real gas, but it frequently provides a useful approximation and furthermore its structure is informative since it springs from a physical view of the behavior of gas molecules.

For such a model there is no need to ask the question "Is the model true?". If "truth" is to be the "whole truth" the answer must be "No". The only question of interest is "Is the model illuminating and useful?"

I decided that I should start with a simple model with many simple assumptions and work from there. If the assumption does not work then I will then make my model more complex. I should keep it simple (stupid!). A programmer should not add functionality until deemed necessary. Always implement things when you actually need them, never when you just foresee that you need them. A famous programmer said that somewhere.

My design principle is, make everything simple if you can get away with it.

Broad Overview of My Pipeline

- At first, start the engine and drive to a sufficient speed

Then, every time the vehicle control is about to run out of something to do:

- Get sensor data about our vehicle and nearby dynamic cars (Where am I? How fast am I going? How fast are nearby cars going? Where are they?)

- Use this data to decide what behavior to do (Should I go left? Go right? Stay on my lane?)
- Based on the decided behavior and the sensor data I've mentioned earlier, we should generate an actual path that would provide a smooth, comfortable and safe ride to accomplish the goal.
- Convert this path into something that the controller understands before sending it to the controller

Data Structures

Ultimately, I ended up making five important data classes to implement my pipeline.

`Vehicle`

- This object is responsible for making sense and storing data about a vehicle, like our vehicle and other vehicles nearby.
- It can answer things such where and and how fast is it? is there a lane at its left? Which lane is in its right? how close are we from vehicle in front of us and how fast is it since we last checked?

`BehaviorPlanner`

- This object is responsible for suggesting which behavior to do based on the information we have about our vehicle and nearby vehicles. For simplicity it returns whether it thinks we should go left, turn right, or stay on its lane.
- NOTE: This is probably not good design practice but as of writing but because it is used in behavior determination, this class contains a function the distance of the closest vehicle in a given lane and position (front or back) given a set of vehicles to a vehicle of interest. As a result it also updates respective members of the vehicle class. (Might inject this out if I have the time)

`JMT`

- JMT stands for "Jerk Minimized Trajectory".
- This object represents a quintic polynomial function of a number which has six coefficients. Once instantiated, you can evaluate this polynomial by giving a value.
- The six coefficients are determined by giving 7 numbers upon instantiation.
- Jerk is the instantaneous change in acceleration over time like acceleration is the instantaneous change in speed over time. They say that people find rides with jerks really uncomfortable and are kinda okay with having acceleration as long as the acceleration is constant or not so high. This is why we want to minimize the jerk of our path. Luckily there's an algorithm to get that. (Thanks math!)
- Suppose that position can be represented by as a function of time, then we know from high school physics that by definition velocity is the derivative of position and acceleration is the derivative of velocity and jerk is the derivative of acceleration.
- Suppose we call a set of position, velocity and acceleration a `state`, and we have our starting state, our desired state, and how much time we have to get from one state to another, there is actually a formula on how to get the polynomial path position function that minimizes the squared jerk to get from the start state to the desired state given the duration. This start state, desired state, and time duration are the seven numbers we give and the resulting polynomial function this object represents is that path position function that minimizes the squared jerk.

`Trajectory`

- This object is responsible to output the actual path along the road that we want the vehicle to take given the desired behavior and current predicted state (Latency considered) of the vehicle. This path is encapsulated by two **`JMT`** objects. One **`JMT`** object represent the path that the vehicle takes with respect to the road moving forward, and the object represents the path the vehicle takes with respect to the road moving sideways.
- To generate the two **`JMT`** objects we must specify a start state, target state and a time duration that this object will calculate based on the information we have about our vehicle in relation to nearby vehicles.

`PathConverter`

- This object has a representation of the global map of the highway. It takes two **`JMT`** objects which represents the path we wish to take along the road sideways and forward. This is something the controller does not understand. The controller understand discrete points along the map in cartesian coordinates.
- So this object is responsible for converting this path into discrete points when we give it the distance between points along the path and number of points.

Deciding Which Behavior

- Given our sensor information, how do we decide whether to turn left, turn right, or stay in lane? This got me thinking, there are many things we can consider. We can consider the speed and acceleration of nearby objects for example. But that's just complicated. Everything must be made simple but not simpler. Think.
- Maybe as a human driver the single most important thing to consider is how much "gap" between us and the vehicle in front of us. As much as possible we want to stay just in one lane and not bump the vehicle in front of us. But if we are going at the required speed but we are already so near the vehicle in front of us then that means we should think about moving left or right. But where should we move? Left or right? Is it even possible to turn anyway? maybe there's just cars in that lane that we wouldn't have space to turn. So we have to consider how much space there is between the nearest vehicle in our desired lane both in front of us and in our back. Also I think we should prioritize being in the middle lane most of the time because it provides more freedom. As a concrete example suppose, I'm in the rightmost lane and the front car is slow, so I want to change lane. I see that the leftmost lane has no cars in it but the middle lane also has heavy traffic. So I can't switch from the right lane to the middle lane, so I'm stuck. This wouldn't happen if I'm in the middle lane.
- Okay, so I know that this is how I think. Basically if we are too near the front vehicle we should consider switching lane. It only makes sense to switch if the gap between the potential leading vehicle is larger than our current leading vehicle. It's only possible to

switch lane if there is sufficient gap between us and the nearest front vehicle and that there is sufficient gap between us and the nearest vehicle at the back in the lane we which to switch to avoid bumping to vehicles. Of course if we can turn both at the left or right we want to choose the one with the larger gaps. Also we'd all things equal, we'd rather stay in the middle lane. We also don't want to switch lanes all the time because that would just be rude and inefficient.

- So how do I translate this to something a computer can understand and process? We can design something like a cost function or equation that considers all of this. What is the cost of staying in the lane, turning left, or turning right? The higher the cost the less we are inclined to do that thing. We choose the lowest cost. It would be best to use a sigmoid or logistic function because it maps values nicely from zero to one. But that's just a little bit too complicated and unintuitive for me. Make everything simple if you can get away with it.
- Okay, so it's impossible to do (there is no road on our right stupid!) Then the cost the highest possible. If the gap between us and the back or front vehicle is below or above a certain threshold then let's just not risk it, make cost the highest possible. The larger the gap, the better. So let's make the gap inversely proportional to the cost. When we are considering turning, we consider the gap at the front and the gap at the back, but the gap at the front is more important. So let's assign add those two components with different weights. For staying in lane, let's just consider the front gap because if the vehicle at our back is behaving properly we shouldn't worry about it. Also this means that given equal front gaps, staying in lane would lead a lower cost because there is no back gap component. Let's test this in our simulator.
- Okay so with some tweaking I realized that I did not consider prioritizing the middle lane. Let's add a factor that rewards turning towards the middle lane, this a fraction less than one that gets multiplied to cost when turning with makes it smaller. Now it's better. I see that it's swerving all the time when the gap between the lanes are all high so that means the differences between the costs are almost negligible. We need to do something about this. Let's add a turn penalty factor to penalize switching turns even more. Still swerving but atleast we are not crashing into anyone. I guess this can be tuned further.

Generating an Actual Path

- Okay so now we know what behavior/action to do. Either go left, go right, or stay in lane. What location exactly along the forward and sideways direction of the road do we want to end up at at how what period of time? (how many seconds) By that time, what speed do we want to have? What acceleration? Oh my god so many choices. Let's make everything simple if we can get away with it. Let's just make the period of time constant. Okay let's try 1 second if it's too fast to switch lane let's try 1.25, 1.5, 1.75 and 2 seconds.
- When changing lane let's just make our goal velocity the same as our starting velocity along the road moving forward. That seems to make sense. Let's make acceleration zero

both along the road and sideways of the road. Once we reach the center of the target road we don't want be moving sideways anymore so no speed components along that direction, no acceleration either. Zero, zero. We know our target location along the side of the road which is the center of the desired lane how about along the direction of the road moving forward? I think it's safe to estimate a desired distance to be the distance taken given our average velocity at that period of time traversed.

- Okay, so i found that 2 seconds is a decent time to switch lane so that's cool. The desired distance traveled formula also works well for our formula.
- When moving straight and there's sufficient gap or if the car in front of us is moving really really fast, we can go as fast as we can which I say a little below the speed limit to be safe. But if we are too near the front vehicle, just to be safe let's just be a little slower than it but if that vehicle is like super slow like a turtle let's not be that slow let's have a minimum threshold of slowness because to be super slow is just too ridiculous.
- Okay so we have our starting state, goal state and duration. That's great! We can use this to generate a path using our `jmt` object I discussed above. Let's also save this target state that we can use as our starting state later.

Converting the Path to what the Controller Understands

- Yay, we have our path now! But this paths are given with respect to the road along the road and sideways of the road. We have to transform it to global cartesian map coordinates with X, Y axes. How to we do this? We are given a sufficient number of waypoints in the X, Y coordinates and we know how they are mapped along the road. We also have a vector with respect to the global X, Y axes pointing outwards the loop.
- We can use a lightweight spline library to estimate a cubic polynomial that passes all the way points with respect to road distance.. We can use this convert any position expressed as a number along the the road and sideways of the road. So given our path function we can generate the X,Y coordinates every 0.02 second increments (which the controller expects) over a span of the traversed time. Let's pass this to our controller! We're done!

Final Statement

- As I've said earlier I think so far this is the most difficult project yet, as it isn't as straightforward as our previous projects. This article is about how I approached this project. It works pretty well but it can always be improved by tuning parameter, designing better cost functions, designing a better estimator of target states, and even considering multiple target states and choosing which is best. There are also assumptions that is based on the simulator given to us that might not be applicable in other cases. For example for the first four seconds I assumed that there are no cars going to block us as I accelerated from a speed of zero to my target speed. I also am waiting to for most of my path points to be executed before appending my new calculated path to be sent. This makes things simpler as given the perfect controller and latency, the state of the starting path is the same as the last desired goal target of the last path. This makes everything simple and works at this case, but will most likely not work at a real environment. Overall, I think this is an interesting project that I loved (and hated!!) working on. My solution is not the best and makes many assumptions not applicable in most other scenarios so take it as a grain of salt!